A boilerplate project, based on a used-in-production web application, for developing and deploying a single-page JavaScript web applications

Kevin Baker
B.A. (Mod) Computer Science
Final Year Project April 2015
Supervisor: Prof. Stephen Barrett

School of Computer Science and Statistics
O'Reilly Institute, Trinity College, Dublin 2, Ireland
I’d like to thank my supervisor Stephen Barrett for being so supportive of this project despite his incredibly busy schedule. I must also thank my classmate Shayan Yousefizadeh for building an incredible iPhone app against the JailbreakHQ API, without his efforts Jailbreak wouldn’t have been such a big success. I must thank and praise the rest of the organising committee for Jailbreak 2015: Ali; Jack; Aisling; Fodhla; Ben; Colm and Vald. Without their incredible effort Jailbreak 2015 would never of raised over €60,000 for charity and I wouldn’t have made JailbreakHQ.org, which spawned the idea behind this project.
Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

_____________________________________       ___________________________
Name       Date

Permission to Lend

I hereby give permission to lend this report to the School of Computer and Statistics Library, Trinity College Dublin.

_____________________________________       ___________________________
Name       Date
Abstract

Web apps like Gmail and Facebook, and native mobile apps present users with a smoother experience than traditional web pages on a daily basis. The traditional model of request and response doesn't suit the interactive web apps that users demand. We explore single-page apps, an alternate model that offers to solve the problems of the traditional model.

This report walks you through this process of building and deploying your own complex single-page app. We illustrate this process with examples from a used-in-production single-page app project called JailbreakHQ.

We outline the differences between traditional web apps and single-page apps by rebuilding two pages of the JailbreakHQ project in the traditional style. We compare the data transfer between client and sever and user load times of the two styles. The results provide useful examples of how the two styles affect these two key web app metrics.

Finally we introduce a boilerplate project for building complex single-page apps. The boilerplate is based on the structure of the JailbreakHQ project. Developers using this boilerplate could skip the complex setup process that we experienced.
Acronyms

- **JS**: JavaScript
- **HTML**: Hyper-Text Markup Language
- **CSS**: Cascading Style Sheets
- **SPA**: Single-Page Application
- **HTTP**: Hyper Text Transfer Protocol
- **AJAX**: Asynchronous JavaScript and XML
- **API**: Application Program Interface
- **URI**: Uniform Resource Identifier
- **DOM**: Document Object Model
- **SASS**: Syntactically Awesome Style Sheets
- **JSON**: JavaScript Object Notation
- **REST**: Representational State Transfer
- **PaaS**: Platform-as-a-Service
- **CDN**: Content Delivery Network
- **CORS**: Cross-Origin Request Sharing
- **HATEOAS**: Hypermedia as the Engine of Application State
- **CI**: Continuous Integration
# Table of Contents

## Part A: Introduction

**Section 1: Introduction**

**Section 2: Single-Page JavaScript Web Apps**
- Traditional Web Applications 3
- Web 2.0 Applications 4
- What is a Single-Page JavaScript Web Application? 5
- AJAX is not Exclusive to Single-Page Apps 8
- Components of a Single-Page App 9

**Section 3: The Single-Page App Philosophy**
- Write-Only DOM 11
- Models Single Source of Truth 11
- State-Driven Views 11
- Views Render a State with a Template 12
- Views Handle DOM Events 12
- Summary 12

**Section 4: The Advantages of Single-Page Apps**
- Separation of Concerns 13
- Multiple Consumers of Data 13
- Multiple Data Sources 14
- Smoother User Experience 14
- Faster Loading Times 15
- Client-Side Internationalisation 17

**Section 5: The Disadvantages of Single-Page Apps**
- Duplication of Work 18
- JavaScript Disabled 18
- Search Engine Optimisation 18
- Reduced Visibility 19

## Part B: Single-Page App Design

**Section 6: Example Project Background** 20
Section 7: Deployment Components

Amazon S3 - Static File Hosting
RESTful API
Cloudflare

Section 8: Frameworks

Backbone.js a Single-Page App Framework
Foundation a Responsive SASS Framework

Section 9: Why RESTful APIs are Appropriate

Stateless Server
Cachable
Uniform Interface

Part C: Implementation

Section 10: Tooling

Managing Dependencies with Bower
Importing JavaScript Code using require.js
Build Tools in Development Mode
The Advantages of using SASS over CSS
The Power of using Jade templates
The Advantages of using CoffeeScript over JavaScript
Mixin Library Extension

Section 11: Developing Single-Page Apps

JavaScript Config

Section 12: Deploying

Two Deployment Components
Deployment Goals
The Deployment Pipeline
Optimising the Deployed Static Files

Section 13: Monitoring

Uptime
Application Errors
Visitor Usage
JS Application Performance
Part A: Introduction

Section 1: Introduction

In this report we make the case for building web applications using the single-page app model. In this model clients load data from an API server using AJAX requests and they create the illusion of navigating through web pages using client-side scripting. In Part A: Introduction, we describe some of the problems presented by traditional web app model where the web server responds with data embedded in HTML. We answer the following questions. How do single-page apps work? What advantages do they have over traditional web apps. How do you build a ‘well-designed’ single-page app?

In Part B: Design, we introduce the example single-page app called JailbreakHQ which has been used by 15,000 users. We use JailbreakHQ to provide illustrate examples of the single-page app approach. We provide some background on the example project and explain why it suited the single-page app design approach. We describe why REST, an architectural style by Roy Fielding[2], is the most appropriate choice of API style if you have the choice.

In Part C: Implementation, we talk about the full lifecycle of developing, deploying and monitoring a single-page app.

Ten years ago, a few hundred lines of JavaScript in one file was common on many websites. In contrast, the JailbreakHQ app has over 50 JavaScript files and 2000 lines of code. We discuss tools for managing this new complexity. This allows client-side code to be modular, re-usable and testable.

We also discuss how using intermediary languages like CoffeeScript, SASS, and Jade as replacements for JavaScript, CSS and HTML respectively can improve the development process. Each of these intermediary languages is pre-compiled to their counterpart in the build phase. We offer how using a third party library called Mixen
can allow you to compose many chunks of re-useable and testable functionality together using CoffeeScript classes. This is not simple in pure JavaScript.

We provide the basis of an deployment system for your single-page app. This system is based on the *JailbreakHQ* deployment process. *JailbreakHQ* has deployed into multiple environments simultaneously over 200 times.

We also consider what and how to monitor your single-page app for uptime, performance, usage and errors. We'll also point out some potential pitfalls that can cause catastrophic errors for a small subset of users.

In *Part D: Analysis of Single-Page Apps*, we outline the differences between traditional web apps and single-page apps using concrete examples. We rebuild two pages of *JailbreakHQ* as traditional web pages. By comparing the two styles we provide you with metrics on how the two designs affect data volume transfer and user load times.

In *Part E: Boilerplate Project*, we demonstrate a boilerplate project for building and deploying single-page apps. We designed it for developers to skip the complex setup process.
Section 2: Single-Page JavaScript Web Apps

There are many ways of building web applications these days but they can be broadly broken up into three categories: the traditional method; the single-page app approach; and the hybrid approach — which is a mix of the two. In this section we outline the problems faced by traditional web apps and explain why single-page JavaScript web apps are suited to fill that void.

It is important to make the distinction between websites and web applications. The page content of web applications can be different for each and every user; there is some sort of dynamic page rendering occurring. The web server is not just serving a static HTML document to the user.

Traditional Web Applications

In the traditional model clients (web browsers) make a HTTP request to a web server. The web server runs some server-side code, maybe requests data from a database, renders that data into a HTML document and returns it to the user. That HTML document frequently references CSS and JavaScript. The client then parses and renders the HTML document, downloads the referenced CSS to style the page and JavaScript to add some dynamic functionality.

This approach is proven and works but it does have some limitations. Let's take for example a web application that lists your phone contact book. You have hundreds of contacts so the web application splits the list of contacts into many pages. If the client wants to see the next page of contacts the client must clear the screen to white and make a new HTTP request to the web server asking for the next page of contacts. The server runs off to the database to fetch the next page of results, renders it as a HTML document, which it returns to the client. The client then parses and renders that HTML document to the user.

The web page listing contacts also contains many components like a header bar, a footer and a side bar containing the list your favourite contacts similar to FIG 2.1. The web server not only needs to fetch new contacts data but re-fetch your favourite
contacts list. The server then re-renders everything again — the two lists, the header bar, the footer — as a HTML document.

This is an obvious inefficiency as we complete redundant work. The client requested a mostly unchanged page. Wouldn’t it be convenient if the client didn’t have to reload the entire web page just one component of it? If the client could ask the server for the data behind the next page of contacts and it rendered it into the page without affecting the other components of the web app.

Web 2.0 Applications

With the advent of Web 2.0 in 2005 [4], browsers started making AJAX [7] requests to the web server for new data which they could update the page. This was the dawn of Rich Internet Applications [4][5]. Sites like Gmail and Google’s search suggestions quickly adopted this AJAX approach as it gave web applications the same rich user experience as desktop apps.

It becomes obvious that this process has some limitations to the standard Web 2.0 approach where the server provides a rendered HTML document and then JavaScript updates it. One major disadvantage is that both the server and the client need to know how to render the data into HTML in the same format for consistency’s
sake. Unfortunately the client and the server probably don’t share a code-base so this rendering process must be programmed in-duplicate on the web server and the client. This is a form of tight coupling between the server-side and client-side code bases. Requiring the client and the server have knowledge of how to present data to the user requires an unnecessary duplication of effort.

Another major flaw of the traditional model and the Web 2.0 model is that they are a bad design model for realtime apps. The request/response model of traditional apps is a non-starter without some whacky browser refresh magic. Web 2.0 apps are forced to store state everywhere: the server, the DOM, JavaScript memory, cookies. This makes building complex realtime apps difficult as sly bugs and inconsistencies can be introduced into the application quite easily. Single-page apps attempt to solve the problems that traditional and Web 2.0 apps suffer from.

What is a Single-Page JavaScript Web Application?

A single-page app (SPA) or sometimes single-page interfaces [4] are a different approach to building web applications. They are best categorised inside the Rich Internet Applications along with Web 2.0 apps. The user experience of single-page apps is similar to Web 2.0 apps but they are based on a fundamentally different architectural model.

Single-page apps (SPA) are made of a four core ideas: client-side rendering; the app is loaded once and never does a full page reload; the app maintains state separate from the server; and a single entry point with HTML5 push state. [5][10] All of the UI is driven client-side scripting using JavaScript.

CLIENT-SIDE RENDERING OF DATA
The first key idea behind a single-page app is that there should be a clear separation of concerns between the server and the client. The client is wholly responsible for rendering data to the user. The server therefore requires no knowledge about how to present data to the user. This removes any coupling between the client and server found in many Web 2.0 applications.
But how does the client know what data resources to request in the first place?
When the client makes its initial HTTP request to the web server it responds with a
HTML document. This document can be as simple as a reference to one JavaScript
file, one CSS style. This document contains no data at all.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="main.css">
    <script src="main.js"></script>
  </head>
</html>
```

The JavaScript file referenced in this HTML document makes AJAX requests to the
web server for data. The JavaScript will parse the responses from the server and
then insert the appropriate HTML into the page to display that data.

As the web server is now performing two very different tasks it can be easier to
separate the server into two components (FIG 2.2). One server serves the static files
(HTML, CSS, JS) and the other is an API server that serves structured data
resources to the clients. For clarity’s sake we’ll refer to them as the static file server
and the API server from now on.
APP IS LOADED ONCE AND THERE ARE NO PAGE REFRESHES
Web users have become accustomed to the concept of web pages. Every page contains different information and you browse the web by surfing from page to page. Using JavaScript, single-page apps can give the user the illusion of a navigating from page to page by re-rendering the content portion of the page without a server roundtrip. Any common portions of the page like a header or footer, and unaffected portions like a sidebar don’t need to be re-drawn. The client only needs to fetch new data from the API server. It doesn’t need to re-fetch or re-parse any HTML, CSS or JavaScript. This gives the user a smoother page transition as only portions of the screen are changing. It should also give the user a faster experience. This can provide a more seamless user experience to users.
SINGLE-POINT OF ENTRY
There is one key property of SPA that is important to point out. They have what is called a single entry point. No matter if the user initially arrives at rte.ie/ or rte.ie/sports or rte.ie/tv the user should always be given the same base HTML file. It is then up to the SPA to determine what content to show to the user based on the initial URL.

HTML5 PUSH STATE
Every page on the internet is located at a unique address. For example, at rte.ie/ you find news, radio and TV schedules. Over on rte.ie/sports you find the sports news. In the case of a SPA if you initially request the home page at rte.ie/ then navigated to the sports news the SPA would redraw the necessary portions of the part for you. If you bookmarked the page or share it with a friend the user would have an out of date URL. That of their entry point — the home page.

That is another problem. The SPA has given the user the illusion of a page change by re-drawing the screen with JavaScript. If the user were to press the back button in their browser the user would go to the previous website not the previous page in our application as we have not been updating their browser history.

The HTML5 Push State API in web browsers however allows JS apps to update the URL and browser history of the client’s browser whenever the SPA gives users the illusion of navigating from page to page. This solves both the bookmarking, sharing and back button issues.

AJAX is not Exclusive to Single-Page Apps
It is important to point out that even though single-page apps request all of their data using AJAX requests after the initial page load containing the HTML/CSS/JS; AJAX requests are not exclusive to SPAs. Rich User Interfaces [4] describes web apps that use AJAX to provide some level of interactivity above traditional web apps. SPAs are the richest subset of the rich user interfaces family.
Components of a Single-Page App

Single-page apps are made up of many components with a specific job — this is to encourage separate of concern allowing you to re-use components in many parts of your application. This is very necessary as your SPAs grow from one or two views (pages) to twenty plus of views.

ROUTER
The router is the centre of any single-page JavaScript app. As SPAs have a single-entry point there must be some logic that decides what data to fetch and views to render based on the URL of the application. That is the router’s job. The router should also keep track of what views are needed and what views to close as the user navigates around the application.

MODELS
Models are object definitions. Using our contact book web app example from earlier we could call a Contact a model. Each model definition has attributes like name, email address, phone number, picture etc. Each contact is an instance of a Model. We store state information inside models and no where else. When attributes in the model are changed the Models emit a change to event to any listeners. Models also handle the syncing of state between the client and the API server.

COLLECTIONS
Collections are groups of Models of the same type. For example, a list of your favourite contacts is a Collection. It is important to point out that collections in your SPA may just be a portion of the overall logical collection as the whole list may be too large to transfer to the client from your API server. For example, your contact book may contain 400 contacts, it would be inefficient to send all that information to the client if they don’t need it.

VIEWS
A view renders one piece of state data to a portion of the screen — it is in essence a component on the screen. Views listen to changes to that state data. On a change that view should re-render it’s portion of the screen to reflect the update in state change. Views render state information into HTML using templates.
Views also listen for DOM events (click, form submission, etc.) and handle those events. A View for example might handle reading values from HTML form inputs when a user clicks the submit button. The view update its model.

Views can also nest other Views inside them. This can be a useful pattern for building complex pages where one page could contain many different pieces of data. Let’s take the contacts example again, the contacts list is one view backed by a collection of contacts, the favourite contacts list could be another view backed by another collection. The two views are independent of each other, they rely on different data. We discuss patterns like this in more depth in Part C: Implementation.

TEMPLATES
Templates are functions that take application state, like a models attributes, as context. The context information is usually in the form of key/value pairs. This context is the template function’s sole parameter. The template function outputs a HTML string. This HTML is then inserted into the DOM by the View that called the template function. By separating your templates into their own files allows designers with very limited code knowledge to edit templates — a big advantage if you have a non-technical designer.
Section 3: The Single-Page App Philosophy

In the previous section we discussed the components that build up a SPA. We particularly highlighted that each component has a separate task in order to separate logic into standalone testable units. In this section we outline the reasons for why those components exists and why they act the way they do.

Write-Only DOM

The DOM is the object model behind a HTML document's structure. JavaScript can be used to interact with the DOM to change any attribute of a HTML element like it’s position or style. You could use the DOM to store state information. This is what many Web 2.0 applications do. They read state from the DOM and then do actions based on that DOM state.

In properly designed SPAs you should only ever write information into the DOM you should never read from the DOM. The reason for this is the next point: models are the single source of truth.

Models Single Source of Truth

State information should be stored in a single location in your application where possible. By storing all state information in models, and not other places like the DOM, we limit the possibility of state information not being updated correctly across all of its local copies. This could quickly might lead to inconsistencies inside our application or a mess of spaghetti code to update the state information in multiple locations.

State-Driven Views

The content of the page should be driven by the application’s state. If we update the state the content of the page should update correspondingly. The cleanest way to implement this goal is for views to be provided with a piece of state like a model. The view then listens to the model for state change events. When the view detects a state change event it can re-draw it’s portion of the screen to the user. This means that the model has absolutely no knowledge of the view or it’s underlying resources.
Views Render a State with a Template

When a view wants to render state information to the user it must structure it inside some HTML and then append that HTML to the DOM. This is done using templates. Templates must be provided with context information. This information comes from the view’s. The context information might be state information like the view’s model’s attributes or the current date and time. This means that the view must have knowledge of what context information is required by the templates. Templates are normally written by developers in a templating language like Jade, not JavaScript.

Views Handle DOM Events

Views should be the only part of the application listening to DOM events. DOM events are events like button clicks, hover events and form submissions — user interactions with the web page. The DOM content is constructed inside the templates. Views are currently the only part of the application that have knowledge of the templates. To listen to DOM events you must have knowledge of the DOM. Requiring only views to listen to DOM events reduces coupling inside your application.

Summary

The philosophy behind SPA design is to make sure every piece of the application has only one concern. Routers handle navigation. Models manage state. Templates render context information into HTML. Views convert application state into HTML for the user. Views also manage interactions from the user. If your models or templates start to grow into monsters of spaghetti code it is probably because you have broken one of the design principles of SPAs. The majority of your code should be inside your views.

Hybrid web apps — where some of the page content is rendered by the server and some is rendered by the client — don’t follow the principles of the SPA philosophy. They typically store state information in the DOM, which can lead to sly bugs and inconsistencies. This is where you can make the clearest distinction between SPAs and hybrid web apps.
Section 4: The Advantages of Single-Page Apps

There are many advantages to using SPAs from a smoother user experience; to the architecture it forces your application stack into; to faster loading times. Here we layout the advantages of single-page apps over traditional web apps.

Separation of Concerns

The SPA design approach forces your architecture into components that can are good at one thing, and one thing only. The API server can now be concerned only with providing data to clients in a standardised format like JSON. The server now has no knowledge of how that information can or should be rendered to the user. In the traditional architecture the server had to be concerned with how the data was rendered to the user as the web server rendered the HTML. This architectural separation also allows you to scale the API server and the static file servers separately.

Multiple Consumers of Data

The first major advantage of the SPA design approach is that it forces you to build an API server as opposed to an application server that renders application data into HTML. In the case of JailbreakHQ there were web clients and native clients — an iPhone app. The iPhone app did not want HTML document responses from the server, it wants data in a structured format like JSON. SPAs also want their data in an easily accessible format like JSON as opposed to HTML document.

SPAs force you to build an API server than can be accessed by any type of client — not just web browsers. This is a big advantage if you want to allow 3rd party integrations or want to build have native mobile apps. O'Reilly [4] discusses how your data becomes remixable when you provide a public API. He suggests that the most successful web services are those that have been easiest to take in new directions unimagined by their creators because of how they disseminate data.
Multiple Data Sources

Another advantage of SPAs over traditional applications is that you can let clients merge data sources from multiple sources/services without proxying that data using your server.

You can pull information from a 3rd party service onto your page and leverage the remixability property that is possible with modern web apps [4]. When using the traditional web application approach your web server must make the requests to the 3rd party service before sending that information onto your client like in FIG 4.1 (above). SPAs can make two separate AJAX requests: one to your API; another to the 3rd party service’s API. This reduces the work load on your server as your server isn’t proxying the 3rd party service to you clients. Note: this only works with public unauthenticated APIs that have permissive CORS headers.

Smoother User Experience

One of the major goals for SPAs from the beginning of their design was that they provide a smoother more seamless experience to users. This can improve complex
non-linear workflows [13]. Single-page apps can also reduce the number of clicks necessary to perform tasks [14].

This is achieved by how SPAs re-render portions of the screen instead of the whole screen. In traditional web applications when the user navigates around the site the screen goes completely blank and all of the screen’s contents are re-drawn. SPAs only re-draw the contents on the screen that need to be re-drawn — not the whole web page. This gives the user a smoother user experience [12, 13]. There are practical examples of this in Section 16: Faster Loading Times and Interactivity.

Faster Loading Times

We propose that faster loading times are an advantage of SPAs. We later provide some real examples to back up this hypothesis in Part D: Analysis of Single-Page Apps. Experimental results have shown a performance increase of 55 to 73% for data transferred over the network [14][15][16], when AJAX is used to conduct partial updates. This reduced data transfer can lead to faster loading times.

In a traditional web app clients cache the page’s CSS and JS — for periods even as long as one year. However, the client cannot cache the HTML document as it contains dynamic data. HTML documents in traditional web apps are typically hundreds of lines long. Both SPAs and traditional apps can cache the JS/CSS files.

SPAs separate data and presentation. This allows them to independently cache them. SPAs can cache the HTML document — even if it is 6 lines — as it contains no dynamic content. SPAs move the presentation data into the JS/CSS files, which are cachable.

But what about the API responses? SPAs can cache data responses from API servers, depending on how transient the underlying data is. By completely separating the data and the presentation SPAs have more fine-grained cache opportunities. In some cases SPAs can cache everything — the HTML document, CSS, JS, and API responses. Taking these caching advantages into account, SPAs should be faster than traditional web apps when the cache is hot.
There are arguments to be made that single-page apps lead to a **perceived** faster loading time for users compared to traditional web servers. There are many stages to loading a web page from the initial request, to downloading the results to the browser displaying them to your page. We address this point in Section 16.

However, with SPAs there are many additional client-side opportunities when loading the page’s full content. The SPA can then render onto the screen static items that have no data dependencies like header bars, or content placeholders while it fetches dynamic data from the API server. This can have big advantages for user perceived performance. We provide examples of this in Section 16.

In FIG 4.2 is example of the placeholders and data dependent-less components that Facebook load onto the screen before they have fetched data from their API servers. This progress encourages users to believe the site is faster than it actually is. It can take 700ms to load your Facebook newsfeed on a good day — yet Facebook rarely seems slow. FIG 4.3 shows the JailbreakHQ home page where some content has been loaded but portions of the page are still being fetched from the API server in the background using AJAX requests. SPAs can start rendering lists of data in any order on the page as the data becomes available; not top to bottom like traditional apps.

**FIG 4.2** Facebook's placeholders and data dependency-less components are rendered to the screen before AJAX responses return. Appendix Fig 4.2 for full-scale image.

**FIG 4.3** JailbreakHQ.org home page. With partial content loaded.
Client-Side Internationalisation

Another frequently over-looked advantage of the SPA architectural style is that the API server does not need to worry about internationalisation. The API server only needs to know how handle different character sets and unicode. Almost all internationalisation logic can be moved to the client-side templating engine [17]. This vastly simplifies the API server. Internationalisation is mostly a problem with rendering data in the appropriate format. Client-side JavaScript libraries like i18n.js¹ and moment-timezone² can handle internationalisation.

¹ i18n-js; https://github.com/fnando/i18n-js
² Moment Timezone; http://momentjs.com/timezone/
Section 5: The Disadvantages of Single-Page Apps

Single-page apps are not a silver-bullet that solve all of our potential problems. They do come with some down sides, which make SPAs unsuitable in some cases.

Duplication of Work

One of the biggest downsides of SPAs is that there is a duplication of code, particularly around models and collections. The modelling code is frequently duplicated across both the API server and the client-side applications. If you don’t need the advantages of SPAs and want to cut down your development time then maybe SPAs are not for you.

JavaScript Disabled

Not everyone online has JavaScript enabled in their browser. Normally this is a security precaution taken by users. Whereas traditional apps can follow an approach of graceful degradation, where the website still works but isn’t as interactive without JS enabled. SPAs do not work at all without JS enabled. In 2010 Yahoo published statistics stating that 2% of US users, 1.29% of UK users and 0.26% of Brazilian had JavaScript disabled[1]. This is a non-trivial number of users.

Search Engine Optimisation

There are some issues surrounding search engine crawlers being able to read the contents of SPA websites. This means that your content will not be indexed — as a result your website would not show up in search results. Google’s search engine crawler has been able to process single-page apps since 2007, however, the crawlers from Bing and Yandex still lack JavaScript functionality.

There are solutions to this problem like providing HTML snapshots or sitemaps to the crawlers that contain your content. These however need to be generated by a server which increases your maintenance costs. There is considerably more work involved in optimising a SPA for search engines then there is in optimising a traditional app.
Reduced Visibility

Another disadvantage of single-page apps is that they can lead to reduced visibility using Fielding’s definition [2], the ability to monitor a component easily. By moving to a fat-client model we have less of the stack to monitor. This means that large portions of our application happen on other people’s machines, which can be hard to measure cheaply and effectively.

In Part D; Analysis of Single-Page Apps we show that measuring load times of single-page apps is tricky because there is no singular the-page-has-loaded event like with traditional web apps. In Section 13: Monitoring we discuss some of the ways to manage this lack of visibility using open source monitoring tools. However, adding these tools to your client is yet another development hurdle that must be factored in when building single-page apps.
Section 6: Example Project Background

It is useful from time-to-time to know the background of the example project JailbreakHQ that was as the basis for the static app boilerplate.

JailbreakHQ is Ireland’s largest student-run charity event, involving over 200 students from 8 Irish universities. Participants in the event attempt to race from a starting location in Dublin to a mystery Location X somewhere around the world without spending any of their own money. The goal was to raise €100,00 and much needed awareness for two charities: Amnesty International and St. Vincent de Paul.

There are a relatively simple list of features. There are two groups of users: anonymous users; and administrators. Anonymous users can browse the teams and check out where they are in the world they are. Anonymous users can also donate to the charities by providing their credit card information. Admins can login to an admin panel and administer aspects of the race from there. More than 50% of JailbreakHQ users are on mobile devices. The website had to be responsive — adapt to best suit the client’s screen size.

This provided the project with some interesting problems to face. Allowing donations through the website required that we implement SSL/TLS everywhere. The admin login feature required authentication. Authentication is an interesting problem for SPAs as you cannot trust the client.

JailbreakHQ has an iPhone for users to follow the goings on. It was clear that we were going to require an API for the iPhone app to fetch data from. It only makes sense that the iPhone app and website use the same API in order to remove any duplication of effort. When you have multiple consumers it makes sense to build one standard API against which you build all your clients, including your website. This was one of the main reasons for using the single-page app approach when designing JailbreakHQ’s website.
Section 7: Deployment Components

We are briefly going to mention all of the components for a SPA. We mention specific services like Heroku, Amazon S3, Cloudflare to give concrete examples but they are only for reference. All of these services are replaceable with alternatives.

Amazon S3 - Static File Hosting

The first component is your static file host. Amazon S3 is a scalable static file storage service. You can put any files you like into storage containers called buckets that can be publicly accessible. We have two different buckets on Amazon S3 in the above diagram. One bucket holds static media files like user avatar uploads, etc. The other bucket contains the optimised HTML, CSS and JS files. This is the same as running an nginx or Apache web server to serve a folder of static files. In Section XI: Deployment we discuss how you can automatically upload the latest files to Amazon S3 on a successful build.
RESTful API

The second component you need to deploy is the RESTful API. *JailbreakHQ* was deployed to Heroku, a low-barrier-to-entry Platform-as-a-Service (PaaS). Deployment to this service is as simple as a git push. This RESTful API is backed by a database and a few third party services but clients have no knowledge of what backs the API only the interface it provides. You need to ensure that your API returns the correct permissive CORS (Cross-Origin Request Sharing) HTTP headers if your API and static files do not share the same origin.

Cloudflare

Cloudflare is a CDN and DDOS-prevention service that works by reverse-proxying your content to users. You host your DNS records on Cloudflare and they insert themselves into the middle of connections with your users.

**CONTENT DELIVERY NETWORK (CDN)**
The first important feature of Cloudflare is that they are a CDN. A CDN is a geographically distributed set of caches. The idea being that if your static files are geographically closer to your users then clients will experience a lower latency and thus faster downloads. Using a CDN in front of S3 is also advised as they charge you GET per-request made, which could get expensive.

**UNIVERSAL SSL**
Cloudflare provides an SSL proxying service called Universal SSL. They create a certificate for your domain for example for *.jailbreakhq.org. When Cloudflare is reverse-proxying the static files in your S3 bucket at jailbreakhq.org.s3…com, which also has an SSL cert, Cloudflare has what they termed full-SSL. SSL connections from the client to them and from them to the origin server. This can be a handy trick of getting SSL on your single-page static app cheaply. A word of warning: Cloudflare are essentially performing a man-in-the-middle attack on you. Cloudflare can read what the user thinks is encrypted traffic.
Section 8: Frameworks

As client-side apps like SPAs grow in complexity there is an ever increasing number of libraries and frameworks being released to deal with the common problems and patterns. For the JailbreakHQ project and the boilerplate project we use these two commonly used frameworks to speed up the development process.

Backbone.js a Single-Page App Framework

Backbone.js gives structure to your web applications to your SPAs by providing models with key-value binding and custom events, collections with a rich API of enumerable functions, views with declarative event handling, and connects it all to your existing API over a RESTful JSON interface — backbonejs.org

The job of Backbone is to provide the components of the SPA that are essential to get your application going. Backbone is used by USA Today, Hulu, Wordpress.com and Stripe to name just a few.

Backbone makes assumes you have a RESTful API structure. If you follow the common RESTful API structure, Backbone’s assumptions will work. By making these assumptions the majority of your models and collections can be quite small. The average length of a model in JailbreakHQ is 6 lines. This reduces one of the disadvantages of SPAs: duplication of model code across client and sever.

Backbone follows our SPA design philosophy whereas other SPA frameworks do not. AngularJS stores state in the DOM. This requires you to write application logic in your templates — this adds coupling to you app. AngularJS is good for Hybrid web apps but poor for pure single-page-apps. Knockout.js completely decouples model state from view state [10]. This is a breach of separation of concerns. Meteor has the problem that is a monolithic framework that needs to control your back-end as well as your front-end [10]. The API and your client should be completely separate. Or you lose the advantages of allowing multiple consumers use your API.

3 BackboneJS Examples; http://backbonejs.org/#examples
Backbone is a very slim framework based on the MVC pattern [42]. This has another advantage, it doesn’t require too much framework-specific knowledge in order to use it unlike Knockout and Meteor, which have large learning curves.

Foundation a Responsive SASS Framework

Foundation 4 is a SASS framework designed for building responsive websites — adapts to best suit the size of the user’s screen. It uses SASS, a pre-compiled CSS language, that allows use of variables, which makes Foundation very customisable. Foundation is used by The Washington Post, PBS and Mozilla [6]. Foundation is designed to be mobile-first and provides a huge list of features for building responsive websites.

Foundation provides a CSS scaffolding for arranging content on the screen and some sensible default styling for HTML elements like forms, buttons, navigation bars. All of these provided components are all customisable using SASS variables. Foundation also provides snippets of JS for standard website interactivity like drop-downs, modal boxes etc.

Section 9: Why RESTful APIs are Appropriate

There are many different architectural styles for building APIs. Roy Fielding’s dissertation [2] contains a great comparison of different styles of networked architectural styles. Here we explain why RESTful APIs are the most appropriate style of API to back your SPA.

RESTful servers optimise for seven properties: performance; scalability; simplicity; modifiability; visibility; portability; and reliability. If you manage the API server these are all properties you want in your system. These properties are achieved by applying six mandatory and one optional constraint to your design: client-server; stateless server; cachable; layered system; code-on demand (optional); and uniform interface.

Three of those are of particular interest to our client-side code: stateless server; cachable; uniform interface. SPAs are automatically client-server so that constraint isn’t particularly interesting.

Stateless Server

Every request from the client must contain all the necessary information for the server to understand the request. The servers must contain no session state. This constraint improves the visibility, reliability and scalability of the API server. [2] This constraints slightly adds more work to our clients — we’re already building a fat client. For example, with authentication the client must send some token to the server on every request. The stateless server constraint is a necessary tax to achieve the seven advantageous properties of REST.

Cachable

This property is a huge advantage to SPAs. Traditional web apps have trouble caching application data as it is embedded in HTML documents, that are themselves difficult to cache. As the data in SPAs comes from a separate source, the API, it is advantageous to cache it where suitable. RESTful API are encouraged to make responses cachable by the client. HTTP caching is implemented by every browser
and if the API returns a response with cachable headers it happens automatically in the browser.

If we examine an alternative API design like RPC or RMI the responses are not cachable as requests in RPC and RMI are about calling functions/methods and not reading/writing object representations like in REST. The increased cachability of REST is one of the reasons REST is great for SPAs.

Uniform Interface

*The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components* — Roy Fielding [2]

There are four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages and hypermedia as the engine of application state. The key abstraction of information in REST is a resource.

**IDENTIFICATION OF RESOURCES**

Resources should be identifiable by some uniform resource identifier. The typical way of fulfilling this constraint on HTTP REST APIs is to use a unique URL per resource. For example: `http://jbapi.net/teams` is a resource representing the collection of team objects. Whereas `http://jbapi.net/teams/66` is a resource representing the team with the id 66. This is great for SPAs as we can easily identify resources both locally and on the remote API. This also increases the cachability of responses as only one resource is at each URL.

**MANIPULATION OF RESOURCES THROUGH REPRESENTATIONS**

Representations can only be changed by providing the new representation. This restriction means you cannot ask the API server to add 1 to a counter. The client must first know the counter value, add one and send that new value to the API server. This is not a problem for SPAs as we have a local copy of the representation...
state in our models and collections. We can make the changes locally and then send the whole new representation to the API. This constraint adds no work to the client.

**SELF-DESCRIPTIVE MESSAGES**
Self-descriptive messages allow the clients to infer the semantics of the interface. Messages are described using media types (XML, JSON) and methods (GET, POST, PUT). This is a big win for client-side code, allowing us to generalise portions of our apps as API resources are consistent. Backbone relies on this property of REST APIs to make large assumptions, which reduces the amount of work developers need to do.

**HYPERMEDIA AS THE ENGINE OF APPLICATION STATE (HATEOAS)**
Interfaces of REST APIs should be navigable by the client. The client should be able to infer the relationship between resources by following hypertext (links / references to URIs) in representations. The goal a navigable API is that clients do not have to rely on out-of-band information — information the API server didn’t tell the client. This property reduces coupling between the server and the client.

The HATEOAS requirement is probably one of the most under-implemented constraints of RESTful APIs. For example, the JailbreakHQ API⁵ doesn’t implement HATEOAS. The reason for this is mainly that it is difficult to implement cleanly inside the API server code — this is a limitation of the tools. It is clear that an API doesn’t have to be 100% compliant in order to achieve the vast majority of the benefits that the REST architecture provides. In the JailbreakHQ project we sacrificed tight coupling between the server and client for reduced development time when building the API server.

---

⁵ JailbreakHQ API Docs; (http://docs.jailbreakapi.apiary.io/)
Section 10: Tooling
As SPAs grow in complexity and size it is important that you are using the right tools to manage the complexity as best you can. All of the tools discussed here are bundled and setup in the Boilerplate project we introduce later in Section 18.

Managing Dependencies with Bower
No developer wishes to re-invent the wheel with every project. As complexity grows all developers want to rely on higher and higher layers of abstraction to build their projects. Complex SPAs are no different. The JailbreakHQ project had 21 direct dependencies and over 30 dependencies in the dependency hierarchy. You need a tool to manage those dependencies in your static app just as you would use Maven or pip when developing Java or Python applications.

Bower is a tool that manages client-side dependencies. Every project specifies their dependencies in a bower.json config file. For example the JailbreakHQ project’s config in FIG 10.1 (next page).

Package names and Github repository URL or version number are listed of the package we wish to include. The backbone project has a dependency on jQuery and underscore.js which are also included in the JailbreakHQ project — though not directly listed. Any project that is on GitHub or in the bower package repository is compatible with bower. That is one of the main reasons for choosing bower over its competitors like Yeoman.

---

6 Bower; http://bower.io
Implementing JavaScript Code using require.js

JavaScript as a language has no concept of file imports or dependencies between scripts [18]. As the complexity of your SPA grows you will want to start separating your components into separate files and separate modules. It would be convenient if JavaScript had an import/include command, that respected nested dependencies.

Require.js is an implementation of the AMD (asynchronous modules directive) specification for loading scripts asynchronously. This is done by wrapping our modules in functions that can get executed once all the necessary dependencies for that module have also been asynchronously loaded.

FIG 10.1 JailbreakHQ bower config file
Another advantage of defining JavaScript modules is that modules only export their interface onto the global namespace. As your application grows in complexity, using require.js modules will keep you from accidentally polluting your JavaScript global namespace.

**COMMONJS**
CommonJS is competing specification for defining and loading JS modules. However, CommonJS loads files synchronously which kills browser performance, whereas require.js loads them asynchronously [18]. CommonJS is best used in projects like Node.js where all the JS modules are stored locally [18]. This is the main reason we choose require.js over a CommonJS implementation like Browserify.

**DEVELOPMENT MODE**
In development mode it is useful to allow require.js load each module as a separate file. This means that any JS errors in your debugger will map back to a line in a real file. However, loading all these files separately requires many HTTP requests, which is slower than one HTTP request for one large JS file. In later sections we discuss using the require.js optimiser to serve one optimised JS bundle in production rather than fifty separate files. The fewer the number of HTTP requests the client must make for JavaScript files the faster the website will load. [20]

**REQUIRE.JS CONFIG FILE**
The core of your JavaScript application is your require.js configuration file. That is where you can specify aliases for modules and for JavaScript modules that don't implement the AMD specification. Backbone.js and underscore.js do not implement the AMD specification — wrap themselves in a function, declare their dependencies and declare what the module exports into the global namespace.

In the JailbreakHQ project we added shim lines to the require.js config to handle this issue. FIG 10.2 (next page) is an example require.js config file. The config specifies that underscore.js exports '_' into the JavaScript namespace; The final six lines list the modules jQuery and underscore as dependencies of Backbone, i.e. the client must load jQuery and underscore before attempting to execute the contents of backbone.
There are many tasks that you will need to repeat frequently when developing your single-page app. Tasks like running unit tests; linting; or compiling intermediary languages like CoffeeScript. During deployment we will want to automate tasks that optimise the static files for improved content delivery [20].

**GRUNT**

Grunt 8 is a popular task runners and it is used by companies like Twitter, Wordpress.com and Mozilla. Grunt can watch for changes to your source files and then it can run tasks based on those changes. For example, in the *JailbreakHQ* project whenever you edit a CoffeeScript file Grunt runs the CoffeeScript compilation task and outputs the corresponding JavaScript source file automatically. Grunt is also incredibly extensible. There are over 4,400 grunt plugins currently available.

---

8 Grunt; http://gruntjs.com
The Advantages of using SASS over CSS

Syntactically Awesome Style Sheets ⁹ (SASS) is a CSS extension language that adds the missing language features that developers have always wanted like: variables; mixins; and nested rules, which makes managing the complexity of large CSS projects easier.

SIMPLER SYNTAX
SASS removes unnecessary syntax there are no semi-colons or brackets but rather indentation is used instead. If you prefer not to be reliant on indentation you can use SCSS, a CSS-like syntax that contains all the features of SASS.

VARIABLES
Variables are the number one missing feature from CSS. With SASS variables you can reference a colour code for example and then by updating the colour code in one location in your code it will update across your entire project.

SASS variables also introduce a whole new level of customisable CSS frameworks. With any package you install you never want to edit the package source files, as this makes upgrading near impossible — with pure CSS frameworks you either had to edit the source or override those CSS properties in your own code, this however means clients must parse more CSS rules. However, if frameworks use SASS variables you can easily customise that framework by specifying values for those variables that override the variable placeholders in the framework code at build time.

MIXINS
Another advantage of SASS is mixins. Mixins are like functions in SASS. In CSS it is quite common to have a different browser-specific style rule for new selectors like border-radius as they make their way through the RFC process. It is faster for developers to write the one line including the border-radius mixins than it is to write out the four rules every time.

⁹ SASS; http://sass-lang.com
NESTED RULES
Nesting styling rules makes it easier to keep your CSS selectors very specific and avoid affecting other parts of your project with vague selectors. Below on the left is the SASS; on the right the outputted CSS.

@ mixin border-radius($radius)
  -webkit-border-radius:  $radius
  -moz-border-radius:     $radius
  -ms-border-radius:      $radius
  border-radius:          $radius

.box
  width: 50%
  background-color: #eee
@ include border-radius(5px)

FIG 10.2 Example of a SASS mixin. It handles the browser-specific CSS prefixes.

NESTED RULES
Nesting styling rules makes it easier to keep your CSS selectors very specific and avoid affecting other parts of your project with vague selectors. Below on the left is the SASS; on the right the outputted CSS.

#main p
  color: #00ff00
.redbox
  background-color: yellow
  color: black

#main p {
  color: #00ff00;
}
#main p .redbox {
  background-color: yellow;
  color: black;

FIG 10.3 SASS syntax showing nesting through indentation

FIG 10.4 The corresponding generated CSS syntax

The Power of using Jade templates

Jade\textsuperscript{10} is the templating engine we used to build JailbreakHQ and it bundled with the boilerplate project. We choose to use a templating engines over writing templates because of the raw power that it provides. Solutions like standard Backbone.js templates do not provide the same power as Jade templates. Jade is a very terse language for expressing templates (example FIG 10.5, next page). Other templating engines such as Handlebars require you to write the HTML tags and placeholders.

\textsuperscript{10} Jade Language Reference http://jade-lang.com/
JADE IS TURING COMPLETE

Jade is turing complete as it can execute any JavaScript function. As a result, it is incredibly powerful. However, with great power comes great responsibility. It can be easy to deeply couple your model, view and template because of this property [17]. A handy rule of thumb we used to avoid this risk when writing templates for JailbreakHQ was that we limited ourselves to calling a small set of functions such as Humanize functions for formatting strings and numbers. We would never call model or view functions from inside the jade templates.

```jade
include mixins.jade

h3.section-header Recent Donations

if error
  +alert("recent donations", errorMessage, errorStatus)
else if loading
  include loading.jade
else
  ul#donations.donations-list
    each donation in donations
      li
        span
          em &euro;
          = Humanize.intComma(donation.amount/100)
          | from #{donation.name}
          if donation.team
            | to
              a(href="/teams/#{donation.team.slug}")#{donation.team.names}
        span.time
          = moment(donation.time*1000).fromNow()
        else
          span There are no donations yet! Why don't you help us out?
            a(href="#").donate-button Donate

  if totalCount > donations.length
    p.and-more and #{totalCount-donations.length} other donations
```

FIG 10.5 The Jade template file that produces the list of donations on the home page of the JailbreakHQ.org website.
INCLUDES
Jade comes with many useful language features that many other templating engines do not come with like including one template inside another. This allows us to create reusable template components. Once such component is the `loading.jade` file, which is used in FIG 10.5. This includes the necessary HTML to present a loading animation to users. This loading animation is used in 7 different templates currently. Include prevent us from repeating ourselves in our code.

 Templating solutions like underscore templates have no concept of this feature. Other templating engines like Handlebars has a similar feature called partials, but it lacks the same expressiveness as Jade, requiring you to write a large amount of boilerplate code.

MIXINS
Mixins allow you to create re-usable blocks of Jade code. One of Jade's core design principles is reusability [24]. Mixins are unique to Jade as far as we can tell. We are unable to find any other templating engine that provides a similar feature.

```jade
mixin errorContainer
  .row
   .small-12.columns
     .error-container
       block

mixin option(value, activeValue, text)
  if value == activeValue
    option(value="#{value}", selected) #{text}
  else
    option(value="#{value}") #{text}
```

FIG 10.6 An example of two Jade mixins inside the JailbreakHQ project

The first mixin uses the keyword block. This allows you to nest contents inside this mixin for example. This is an incredibly valuable feature of Jade templates that fosters reusability.

```jade
+mixin errorContainer
  h3 404 Page Not Found
```
TEMPLATES TO HTML DOCUMENTS
There are two methods for using Jade templates. You can compile Jade templates to HTML documents or to JavaScript functions. The base HTML document in the JailbreakHQ project is actually a Jade template that is compiled to a HTML document at build time. This allows JailbreakHQ to use Jade language features like mixins and includes but also inheritance, which we haven't discussed.

TEMPLATES ON CLIENT
The most powerful feature of the Jade template engine is the idea of client-side templating. The idea is that during the build phase we compile the Jade templates down to JavaScript functions. These JavaScript functions take one argument, the templates context with which they interpolate a string of HTML tags and content.

The client downloads the JavaScript functions and a small JavaScript jade runtime; about 8KB uncompressed. The client can then provide the context, usually a JavaScript object and execute the function to be provided with a string of HTML. This string of HTML can then easily be inserted into the DOM using jQuery.

Unlike with traditional apps where all of the rendering is done when the page is first loaded, single-page apps can render content to the screen at any time. As a result they need to be able to execute templates on-demand.

The Advantages of using CoffeeScript over JavaScript
JavaScript is not the winner of any language design competitions. It has many problems including overloading of the + operator with type coercion, the error-prone with statement, strict reserved word policies, it is build heavily against the DOM API which is also poorly designed, strange equality handling rules, etc. [24, 25, 26, 28].
JavaScript is however the best choice of scripting language for modern browsers. JavaScript as it is fast [29] and widely supported by browsers.

CoffeeScript attempts to solve some of the problems of the JavaScript language. CoffeeScript is a programming language that compiles down to JavaScript code. CoffeeScript is designed in the vein of Python or Ruby, so it is a very familiar syntax for those developers. CoffeeScript comes with many features like fat arrows,
classes, the super keyword, the existential operator and more that are designed to fix the common problems JavaScript developers face.

**FAT ARROW (=>)**
One of the common problems faced by JavaScript developers is the use of the keyword ‘this’. This is due to the rather complex scoping rules in JavaScript [28, 30]. There are even behaviour differences for the keyword this between strict and non-strict mode [31]. The ‘this’ keyword has a large series of long StackOverflow answers too [32]. The fat arrow in CoffeeScript handles the common pattern in JavaScript code of juggling the this keyword in anonymous functions — which are incredibly common in JavaScript.

**CLASSES**
JavaScript doesn’t implement classes in the traditional sense like many other programming languages [25, 33]. It does however implement prototypal inheritance, which has more expressive power than classical inheritance [27]. You can implement classical inheritance in JavaScript but it is advised against. This is because objects in JavaScript are so flexible, you will want to think differently about class hierarchies. Deep hierarchies are inappropriate. Shallow hierarchies are efficient and expressive [33].

Prototypal inheritance is a model unfamiliar to many programmers however. CoffeeScript provides classes with classical inheritance as a language feature. This allows developers with a background in programming languages like Java, Python, Ruby, or PHP, which all have classical inheritance to quickly leverage inheritance in a JavaScript setting without learning the complex nature of dealing with JavaScript prototypes.

**SUPER**
Another useful feature of CoffeeScript is the added super keyword. As CoffeeScript introduced classes it also introduces complementary features to better manage your inheritance like super. The super keyword is also important for the Mixin library, which we mention later in this section.
A WORD OF WARNING ABOUT COFFEESCRIPT
There are some disadvantages to using CoffeeScript; the most obvious one is that debugging can become more difficult. On the odd occasion CoffeeScript's sparse syntax can lead to sly bugs, which can require developers to open the compiled JavaScript file and investigate. Without a solid working knowledge of JavaScript you may struggle to use CoffeeScript effectively.

Using source maps, which are supported in Chrome 30 and above, can help aid the debugging process for CoffeeScript 11.

Mixin Library Extension
There is an incredibly useful library called Mixen12 developed by digital marketing company HubSpot. Mixen lets you combine classes on the fly. With it you can build smaller, easier to understand and more testable components, and more easily share code with others. It does not just merge the JavaScript prototypes.

Mixen is bundled into the single-page app boilerplate project presented in Section 17. In Section 19 we provide illustrated examples of the powerful patterns you can compose using Mixen.

12 HubSpot's Mixen Library; http://github.hubspot.com/mixen/
Section 11: Developing Single-Page Apps

Later in Section 18 we describe how to get started building a single-page app using the boilerplate project provided in this project. In Section 19 we describe useful patterns for building single-page apps using the tools and frameworks provided. In this section we provide some advice to keep in mind when developing single-page apps.

JavaScript Config

The JavaScript config is a where we store configuration for our single-page application. There is much debate over how to best achieve a flexible and robust JavaScript config. Joretag in Human JavaScript [10] suggests a method for client getting the config values from the web server using a cookie. This approach requires you to setup more infrastructure and complicates the deployment layout we discussed in Section 7.

The method used by the boilerplate project and the JailbreakHQ example project is the simplest possible. We create a JavaScript object on the global name space. In the case of JailbreakHQ the object is called ‘jailbreak’. These values are hardcoded into the base HTML document. It is important that these values are setup before the main JavaScript file is downloaded and executed.

The goal of your JavaScript config is to keep config values that might differ from environment (local, testing, production etc.) in one place. For example when developing the JailbreakHQ project locally the config value for the API server host was set to http://localhost:8090, where I was running my local API server. In production however that value needs to point on the actual API server, and was set to https://jbapi.net.

We highly recommend that you never mention environment names in your code. That you only reference config values. The reason for this is best described by the 12 Factor App principles [34].
Another aspect of config management is grouping. Sometimes apps batch config into named groups (often called “environments”) named after specific deploys, such as the development, test, and production environments in Rails. This method does not scale cleanly: as more deploys of the app are created, new environment names are necessary, such as staging or qa. As the project grows further, developers may add their own special environments like joes-staging, resulting in a combinatorial explosion of config which makes managing deploys of the app very brittle.

Config values are fully orthogonal to other config values but when they are grouped you loose this property. Managing your JavaScript config carefully using these guidelines is a good way keeping your application in a deployable state.
Section 12: Deploying

In this section we discuss our deployment goals, how deployment can be done automatically, and all without managing any of your own deployment infrastructure. This is the same deployment process that is bundled into the boilerplate project and is based on the JailbreakHQ deployment process that was used to deploy into multiple environments over 200 times.

Two Deployment Components

It is important to point out that in the single-page app architecture, the client-side single-page app code and the server-side API code can be deployed as two separate components. We can deploy the two components on different infrastructure components at different times. As long as the interface between the server and client remains unbroken you could deploy the client-side app hundreds of times without having to re-deploy the API server.

Deployment Goals

There were two primary deployment goals: deploys should be automatic on a git push and there should be almost no deployment infrastructure to manage.

There are a few reasons why we wanted to make deploys automatic on a git push. The idea being that when a developers pushes code to the “production” branch on GitHub there is an automatic deploy of that code to the production severs. This is an approach used by PaaS company Heroku and is one of their key selling points.

The first advantage of automatic deploys is that they are incredibly simple for developers. A manual deployment process can lead to human-error, which can cause downtime for end-users. An automatic process eliminates this risk for developers as automatic processes don’t forget steps. This risk does grow if you deploy to a distributed system. Developers also deploy more regularly when there is no manual process to go through [35]. Overall an automatic deployment process leads to more frequent and successful deploys. This automatic deployment process is referred to as Continuous Delivery [37].
An automatic deployment pipeline also leads to more frequent deploys. Developers can deploy before they have finished developing the entire website [37]. This allows you to iterate many times on your site with improvements and bug fixes. Iteration is a key tenant of the Agile development methodology [38] that was used when building the JailbreakHQ project.

Deployment infrastructure like build servers and deployment registry can be complicated and time-consuming to setup and manage. Since 2011 there has been a service called TravisCI 13 which provides free build and continuous integration services to open source projects. There are alternatives such as CircleCI 14. This approach of relying on a CI service provides all the benefits of CI such as allowing teams develop cohesive software more rapidly [36].

The Deployment Pipeline

The deployment pipeline process is kicked off when a developer merges code into a known branch on a source repository like GitHub. This branch could be named “production” or “testing” and each branch should reflect the state of code behind each deployed environment.

---

13 TravisCI; https://travis-ci.org/
14 CircleCI; https://circleci.com/
TravisCI monitors the GitHub repository branch for changes. When it notices a change it kicks off the build, test and deploy process. If the build or test processes fail then the deployment is cancelled and developers are notified. In the *JailbreakHQ* the source repository contains only the intermediary languages like CoffeeScript and SASS and not the compiled forms like JavaScript and CSS. The first phase is to run Grunt tasks that compile the intermediary languages. TravisCI uses the same Grunt config that we used in development. We then run more Grunt tasks that optimise the JavaScript and CSS files. We discuss how and why this happens in the next subsection.

TravisCI is aware which environment it is attempting to deploy into because of the GitHub branch it attempting to build and deploy. In Section 11 we discussed the JavaScript config. In FIG 12.1 is an example of the JavaScript config with the different environments enumerated. Each environment is nested inside a named +targethtml Jade mixin. This mixin inserts special HTML comments that another Grunt task uses to only include the JavaScript config for that environment.

```javascript
script.
  var jailbreak = {};
+targethtml('dev')
  script.
    jailbreak.url = 'local.jailbreakhq.org';
    jailbreak.api_host = 'http://localhost:8090';
    jailbreak.ga_id = 'UA-58463386-2';
+targethtml('qa')
  script.
    jailbreak.url = 'qa.jailbreakhq.org';
    jailbreak.api_host = 'https://qa.jbapi.net';
    jailbreak.ga_id = 'UA-58463386-2';
+targethtml('prod')
  script.
    jailbreak.url = 'jailbreakhq.org';
    jailbreak.api_host = 'https://jbapi.net';
    jailbreak.ga_id = 'UA-58463386-1';
```

FIG 12.1 JS Config in the base Jade Template using the +targethtml mixin
After building and optimising the code it is bundled as a release. This release needs to be deployed to a static file server. In the case of JailbreakHQ case we deployed the files to Amazon S3. TravisCI provides tools for easily uploading files to an Amazon S3 bucket.

One of the features that was planned for the boilerplate but is yet unimplemented is deployment rollbacks, i.e. if the changes introduce bugs/errors you can instantly rollback to the last good release [37]. For this reason each release of the JailbreakHQ project and boilerplate app are stored in separate numbered directories inside the Amazon S3 Bucket.

Each release folder contains the optimised JavaScript and CSS files, images and base HTML document. If we are deploying into production, the deployment process copies the base HTML document from that new release folder and overrides the prod.html document in the root of the S3 bucket. This prod.html HTML document contains references to the release folder it was copied out of. The production environment nginx reverse-proxy references that prod.html at all times. The QA (testing) environment nginx reverse-proxy always references the qa.html document. We update the contents of those base HTML documents in order to update to a new release or rollback to an old release.

Optimising the Deployed Static Files

In development we wanted to serve ourselves the easiest files to debug. In a deployed environment we want to serve end-users with the most efficient files possible. As part of the deployment pipeline must optimise the JavaScript and CSS files to be as efficient as possible. We run Grunt tasks on TravisCI that: reduce the number of files and reduce the size of the files.

REDUCING THE NUMBER OF HTTP REQUESTS
The first optimisation is to reduce the number of HTTP requests [39]. In development we serve developers all of the JavaScript files separately. Each of these requires an it's own HTTP request.

As we are using require.js to manage our JavaScript dependencies we need to use r.js \(^{15}\), the require.js optimiser. This optimiser merges all of your source JavaScript files into one singular JavaScript file to reduce the number of HTTP requests clients make.

It is suspected that as we move to HTTP/2 this process may in fact reduce performance instead of increase it. This is because of the new multiplexing and the use of a single TCP connection per-origin introduced in the HTTP/2 specification [40, 41]. When we move to HTTP/2 we can leverage require.js AMD’s specification to actually load JavaScript modules asynchronously as required instead of using r.js to compress the modules down into one file. This could solve the growing JavaScript bundle size problem we point out later in Section 15.

**MINIFICATION AND OBFUSCATION**
The next optimisation is to reduce the file sizes of the CSS and JavaScript files where possible. As this code is downloaded by the user the smaller we can make it the faster the site will load for end-users. Minification removes all whitespace, unnecessary trailing semi-colons and comments from CSS. These are useful in development but are unnecessary in a deployed environment. Obfuscation removes all JavaScript whitespace and comments but also renames functions and variable names to shorter equivalents such as \(a, b\) etc.

---

\(^{15}\) Require.js Optimiser; http://requirejs.org/docs/optimization.html
Section 13: Monitoring

In this section we discuss the tools and services you can use to monitor your single-page apps for uptime, errors, performance and usage. You should provide similar monitoring for your API as your single-page app is dependent on it returning responses but that is outside the scope of this report.

Uptime

Uptime is an important measure for your website. The easiest way to measure your uptime, and more importantly get notified when there are issues is to use service like Pingdom. Pingdom is easily replaceable with open source tooling, but you then need a monitoring tool for your monitoring tool, which is why we recommend free services like this for reasonably small-scale projects.

Application Errors

Logging application errors to your server is a vital part of moving to a single-page app. As your application is running client-side any errors need to be reported back to your monitoring service using AJAX requests. This is a fundamental shift from the traditional approach where you could just monitor the application log files on your server.

JavaScript application errors inside your Backbone router, for example, can be catastrophic and can prevent any of the page rendering to your users. It is important that you can find these errors quickly.

We highly recommend the service Sentry or its open source project. Sentry has clients for many programming languages not just JavaScript. Sentry is the error monitoring tool used for both the JailbreakHQ single-page app and the Java API server.

---

16 Pingdom Free Tier; https://www.pingdom.com/free/
17 Sentry Service; https://getsentry.com/welcome/
18 Sentry Source; https://github.com/getsentry
Sentry’s JavaScript client records not only what the JavaScript error was, including it’s stack trace etc. Sentry for JavaScript also records information about the client like their operating system, browser version, IP address etc.

Visitor Usage

Every website wants to know statistics about usage. How many pages did users visit on average? Which pages? How long did they visit for? It is important to point out that you cannot just drop Google Analytics or a similar analytics tool into a single-page app. Google Analytics by default only reports statistics after the initial page load by default. This is no good for single-page apps where navigation between pages can happen without another page load.

This problem is solvable by using the JavaScript client that Google Analytics provides. Whenever the client navigates from route to route you can ping the Google Analytics service to report a new page view. This requires you to load the Google Analytics file using require.js inside your project.

GOOGLE ANALYTICS AND PRIVACY EXTENSIONS

One gotcha you need to be aware of when including third-party analytics tracking code using require.js is that tools like adblockers and ghostery will block requests to them. This can cause catastrophic JavaScript errors that prevent your app from rendering anything to the screen. You can use the approach taken in the JailbreakHQ project.

JS Application Performance

Again as we have moved our web application client-side we need to measure it for performance just like we would any web server application. The best tool we found for monitoring JavaScript app performance was Bucky. It is an open source project.

---

19 analytics.js; https://developers.google.com/analytics/devguides/collection/analyticsjs/

20 JailbreakHQ Google Analytics Error Handling; https://github.com/jailbreakhq/public-static/blob/master/static/src/scripts/app/AppRouter.coffee#L28-34

21 Bucky; http://github.hubspot.com/bucky/
that records the W3C Navigation Timing events [3] and AJAX timing events, and reports them to your monitoring service.

Developer Notifications

Monitoring tools are useful if they provide notifications when something is going wrong. A handy tool is Slack, a real-time team messaging platform. JailbreakHQ hooked up Pingdom, Sentry, New Relic and TravisCI build updates to provide notifications to its developers. This is a great way to increase visibility into your system.

FIG 13.1 Example notifications in the JailbreakHQ dev team Slack channel.

22 Slack; https://slack.com/
Part D: Analysis of Single Page Apps

We performed some high-level analysis comparing single page apps to their traditional web server counterparts. We compared the volume of data transferred between the server and client and the time it takes to load and render the page. These tests are not designed to provide rigorous data on the advantages of SPAs over traditional apps; that is far outside the scope of this project. The tests do however illustrate the fundamental differences between the two architectures by providing concrete examples.

Section 14: Test Methodology

Test Projects

In order to compare and contrast a single-page app and a traditional application we need two web apps that are functionally the same but built in the two different architectural styles. We took the JailbreakHQ example project and built two of the pages using the traditional web application approach where the server renders the HTML. The two test pages were the home page and the teams listings page.

The home page consists of an interactive map with a marker for each team and the start and finish points of the JailbreakHQ race; a list of recent events; a list of recent donations. The home page is interesting because it renders information from many different data sources (teams, events, donations).

The teams listing page contains a list of 20 teams. Users can filter the teams by university and sort them by the amount of money they’ve raised, their position in the race etc. On the single-page app users can load more teams by clicking a “load more” button and more teams are appended to the list of teams. In the traditional web application, the user is presented with standard pagination buttons of next/previous. These two styles are common for their respective architectural models.
This is an interesting test page because of how possible user interactions after the initial page load like filtering and pagination can affect load times of the two pages.

The traditional web application is created by adding the FreeMarker HTML rendering engine to the Java API server. This HTML rendering layer replaces the JSON rendering layer of the API server.

The source code for the three separate test components: the API server 23, the static files for the traditional app 24 and the test single-page app 25 are publicly available on GitHub under the MIT License.

Test Method

The tests were performed using a tool called WebPageTest 26. It is developed by Google and designed to record and measure every detail of a page load, including timing events, number and size of requests, and the rendering process, and Speed Index [5] a measure of perceived load time.

*The Speed Index is the average time at which visible parts of the page are displayed. It is expressed in milliseconds and dependent on size of the view port. It is particularly useful for comparing experiences of pages against each other*

The numbers, visual progress graphs and filmstrips in the following sections are the median result of 18 test runs against each of the two test pages against the two test projects. Each test was performed by an test agent running on a server located in Dublin; the test sites were also situated on Dublin servers. The test runner rendered the page using the Chrome 41.0 rendering engine at desktop screen sizes and its connection was limited to 5Mbps. It should be clear from these test parameters are not rigours.

23 Test API Source; https://github.com/jailbreakhq/JailbreakApi/tree/html-views
24 Test Traditional App Source; https://github.com/Specialkbyte/traditional-static
26 WebPage Test; http://www.webpagetest.org/
Section 15: Volume of Data Transfer

Single-page apps can contain vastly more JavaScript than traditional web apps. We wanted to measure how much extra JS data is there.

Home Page

INITIAL PAGE LOAD
On an uncached page load, the traditional app loads 233KB of JS, whereas the SPA loads 288KB of JS from the server. An increase of 23%. However, if you compare the total amount of all data loaded by the client in both the traditional and SPA cases then it is clear that the largest contributor to data volume downloaded for this test page is not JS code but image data.

<table>
<thead>
<tr>
<th>MIME</th>
<th>Traditional App</th>
<th>% of Total</th>
<th>KB</th>
<th>Single Page App</th>
<th>% of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>4241.3</td>
<td>92.47%</td>
<td>3714</td>
<td>89.11%</td>
<td></td>
</tr>
<tr>
<td>JS</td>
<td>233.2</td>
<td>5.08%</td>
<td>288.1</td>
<td>6.91%</td>
<td></td>
</tr>
<tr>
<td>Font</td>
<td>80.8</td>
<td>1.76%</td>
<td>81</td>
<td>1.94%</td>
<td></td>
</tr>
<tr>
<td>CSS</td>
<td>24.8</td>
<td>0.54%</td>
<td>24.8</td>
<td>0.59%</td>
<td></td>
</tr>
<tr>
<td>HTML</td>
<td>6.79</td>
<td>0.15%</td>
<td>1.8</td>
<td>0.04%</td>
<td></td>
</tr>
<tr>
<td>AJAX</td>
<td>0</td>
<td>0%</td>
<td>58.4</td>
<td>1.4%</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>458.7</td>
<td>416.8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Images are over 89% of the volume of data loaded in each case. In the grand scheme of things, the additional JavaScript overhead of SPAs does not majorly add to data overhead. It is important to point out that for both styles the JavaScript code is served off a CDN and it is cached by the client after the first request, which means the minor increase in data volume is only felt by the client on the first page load and not subsequent ones. Butkiewicz et al [21] find image data volume is upwards of 35% of the total data volume transferred on hundreds of sites across the internet.

As the size of your single-page app grows in complexity so too can the size of the JavaScript code bundle. As an example, the full JailbreakHQ project with all it’s
many views, authentication and extra functionality has a JavaScript bundle of 83KB. This poor scaling can be solved by leveraging AMD (asynchronous module directives) to load JavaScript files as necessary instead of one large bundle. There is a fine balance between the number of JavaScript bundles to produce versus load initial load times. It is however unique for every website and outside the scope of this project.

**cached page load**
On a cached page load, the traditional client only requires to download the 6.6KB of HTML as all the static files (JS/CSS) have been cached. The SPA client has to request no new data. This is because the HTML, CSS, JS data have all been cached on the previous request. The API responses are also all cached. The client renders the page based-solely on locally cached data. This is an example of SPAs separation of data and presentation, which leads to increased cache granularity allowing clients to cache a larger percentage of the data volume — in this case 100%.

**teams listing page**

<table>
<thead>
<tr>
<th>MIME</th>
<th>Traditional App</th>
<th>% of Total</th>
<th>Single Page App</th>
<th>% of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>2226.4</td>
<td>94.49</td>
<td>2225.7</td>
<td>91.66</td>
</tr>
<tr>
<td>Font</td>
<td>66</td>
<td>2.8</td>
<td>66</td>
<td>2.72</td>
</tr>
<tr>
<td>JS</td>
<td>37.3</td>
<td>1.58</td>
<td>97</td>
<td>3.99</td>
</tr>
<tr>
<td>CSS</td>
<td>23.4</td>
<td>0.99</td>
<td>23.1</td>
<td>0.95</td>
</tr>
<tr>
<td>HTML</td>
<td>3.1</td>
<td>0.13</td>
<td>1.8</td>
<td>0.07</td>
</tr>
<tr>
<td>AJAX</td>
<td>0</td>
<td>0</td>
<td>14.6</td>
<td>0.6</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2356.2</strong></td>
<td></td>
<td><strong>2428.2</strong></td>
<td></td>
</tr>
</tbody>
</table>

There is a larger volume of JS data transferred to the SPA client on the initial uncached page load. However, this increased volume is only a small percentage of the total volume of data being downloaded by the client.
White [14], Smullen et al [15], and Merrill [16] all show that AJAX based applications like single-page apps transfer between 55-73% less data over the network than their traditional counterparts when you consider partial updates as well as full page loads.

Summary

It should be clear from these numbers that even with the SPA's increased JavaScript payload there is not a significant increase in total data volume. Data volume increases will also only affect the first page load for both the traditional and SPA designs as they can both rely on CDN's to serve cachable copies of their static files.

QUICK NOTE

In these tests it became clear that quite a large amount of JSON data is downloaded by the client from the API server in the SPA design when compared to the byte size of the HTML files served by the traditional web app. This is mostly due to the design of the Team resource on the API, which includes a large team description field — sometimes over 5000 bytes of data which is not rendered to the user.
Section 16: Faster Loading Times and Interactivity

What are we Measuring?

The process that browsers go through to request, download and render web pages is not simple. Fig 16.1 is the W3C specification of how browsers should do this process [3].

Important things to note are that: the browsers start rendering the page to the user even before it has downloaded the entire web page contents at `domLoading`; the JavaScript that powers single-page apps does not start executing until `DOMContentLoaded` event has been reached.

SPAs attempt to shorten the steps before the `DOMContentLoaded` event as much as possible through simplicity of the HTML document. If the single-page app HTML contains very little content then an single-page app might on average beat a traditional app, with a complex nested DOM structures faster, to the `DOMContentLoaded` event.

Part D: Analysis of Single Page Apps
We proposed in *Part A: Introduction* that SPAs have faster loading times than traditional web apps. This statement is unhelpfully vague. In the following tests we compare the following metrics: start render time; speed index; time to document complete; time to fully loaded for the two web app styles.

The metrics time to document complete and time to fully loaded measure the actual load times. It is important to point out that document complete time for the two styles means something different entirely. Feng [8] illustrates that document complete time for AJAX based applications and traditional apps are not easily comparable measures. This is the reason for the introduction of measures like Speed Index [7], Page Phase Test [8], and User Ready Time [9]. In these tests we rely on Speed Index as it has freely available tools for measuring it.

The metrics start render time and speed index measure the perceived load time. It is important to measure actual download times and perceived load times in to compare the two differing styles. User perceived load times are closely tied to the concept of *User Interactivity and Usability* outlined by Folmer [11] and Teo et al [12]. McMillan et al [22] also closely links user perceived load times and interactivity, though it does point out that *Interactivity* is an undefined or under defined term. Hanssen et al [23] discuss how responsiveness, which they relate with nature of feedback and responses times, is key to measuring *Interactivity*.

**DOCUMENT COMPLETE TIME**
The Load Time is measured as the time from the start of the initial navigation until the beginning of the window load event. This is when JavaScript code can start to execute.

**FULLY LOADED TIME**
The Fully Loaded time is measured as the time from the start of the initial navigation until there was 2 seconds of no network activity after Document Complete. This will usually include any activity that is triggered by javascript after the main page loads.
START RENDER TIME
The Start Render time is measured as the time from the start of the initial navigation until the first non-white content is painted to the browser display. Start render time is an effective measure of user latency perceived.

VISUALLY COMPLETE TIME
The Visually Complete time is measured as the time from the start of the initial navigation until the last piece of content is painted to the browser display.

Initial Loading Times
These tests are what a typical user might expect on their first ever connection to your website. The user will have nothing cached locally and must download everything. This is of particular interest because the SPA cannot do any useful work until after it has downloaded the JavaScript code. Whereas the traditional app can start rendering useful information to the screen as soon as it receives it from the web server.

HOME PAGE

<table>
<thead>
<tr>
<th></th>
<th>Doc Complete Time (ms)</th>
<th>Fully Loaded Time (ms)</th>
<th>Start Render (ms)</th>
<th>Visually Complete (ms)</th>
<th>Speed Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>7675</td>
<td>12372</td>
<td>1891</td>
<td>11000</td>
<td>8078</td>
</tr>
<tr>
<td>SPA</td>
<td>8188</td>
<td>8235</td>
<td>1700</td>
<td>8100</td>
<td>5799</td>
</tr>
<tr>
<td>Difference</td>
<td>513</td>
<td>-4137</td>
<td>-191</td>
<td>-2900</td>
<td>-2279</td>
</tr>
</tbody>
</table>

FIG 16.3 Graph of visual rendering progress of the SPA versus traditional app for an uncached home page load. (Appendix 16.3 contains full-scale image)
Appendix FIG 16.2 is a filmstrip that shows the rendering process of the SPA versus the traditional app. The SPA does perform better than the traditional app in this test case but that is mostly because the map on this page is dependent on JavaScript execution.

**TEAMS PAGE**

<table>
<thead>
<tr>
<th></th>
<th>Doc Complete Time (ms)</th>
<th>Fully Loaded Time (ms)</th>
<th>Start Render (ms)</th>
<th>Visually Complete (ms)</th>
<th>Speed Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>4552</td>
<td>4594</td>
<td>1190</td>
<td>2200</td>
<td>1333</td>
</tr>
<tr>
<td>SPA</td>
<td>1577</td>
<td>5812</td>
<td>1488</td>
<td>3300</td>
<td>2065</td>
</tr>
<tr>
<td>Difference</td>
<td>-3178</td>
<td>1218</td>
<td>298</td>
<td>1100</td>
<td>732</td>
</tr>
</tbody>
</table>

From this data it is clear that the teams page loads faster on the traditional app than the SPA. The traditional app is visually complete a whole 1.1 seconds before the SPA app, 50% faster.

FIG 16.5 is a filmstrip of the rendering progress with frames every 0.5 seconds. In it interesting to look at how the SPA renders the page in this filmstrip. It renders the website header and filter bar to the screen as they have no data dependencies while it awaits a response from the API server with the list of teams to render.

**FIG 16.5** Filmstrip of SPA app versus traditional app for uncached teams listing page load. Frames every 500ms. Top row Traditional app. Bottom row SPA. (Appendix Fig 16.5 shows full-scale image)

When the cache is cold the single-page app is considerable slower than the traditional web app in both the actual load time and perceived load time metrics on the teams listing page. In both cases the traditional app was faster at rendering anything to the screen (first render). These results cannot be surprising as the client...
in the single-page app case must download all JavaScript before any useful work can be done. However, it is important to point out that this case only occurs on the initial page load as the JavaScript is cached for a long period after that point. In a web application with high interactivity this case would be quite uncommon.

**FIG 16.6** Graph of visual rendering progress of the SPA versus traditional app for an uncached teams listing page load. (Appendix Fig 16.6 shows the full-scale image)
Cached Loading Times

This is test case that what would be typical for the second page request or navigation in a traditional web app or an manual web page refresh in an SPA. As SPA navigation is handled internally these loading times are not accurate for SPA navigation, we address those in the next section.

### HOME PAGE

<table>
<thead>
<tr>
<th></th>
<th>Doc Complete Time (ms)</th>
<th>Fully Loaded Time (ms)</th>
<th>Start Render (ms)</th>
<th>Visually Complete (ms)</th>
<th>Speed Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>2402</td>
<td>5304</td>
<td>2387</td>
<td>5000</td>
<td>3941</td>
</tr>
<tr>
<td>SPA</td>
<td>4725</td>
<td>6282</td>
<td>2077</td>
<td>5800</td>
<td>4535</td>
</tr>
<tr>
<td>Difference</td>
<td>2323</td>
<td>978</td>
<td>-310</td>
<td>800</td>
<td>594</td>
</tr>
</tbody>
</table>

Appendix Fig 16.8, Fig 16.9, Fig 16.10 show the home page cached load filmstrip, visual progress and timing events graph respectively.

### TEAMS PAGE

<table>
<thead>
<tr>
<th></th>
<th>Doc Complete Time (ms)</th>
<th>Fully Loaded Time (ms)</th>
<th>Start Render (ms)</th>
<th>Visually Complete (ms)</th>
<th>Speed Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>1152</td>
<td>1181</td>
<td>1197</td>
<td>1400</td>
<td>1208</td>
</tr>
<tr>
<td>SPA</td>
<td>275</td>
<td>834</td>
<td>38</td>
<td>1100</td>
<td>644</td>
</tr>
<tr>
<td>Difference</td>
<td>-877</td>
<td>-347</td>
<td>-1159</td>
<td>-300</td>
<td>-564</td>
</tr>
</tbody>
</table>

In both of these test cases the SPA started rendering to the screen faster than the traditional app. The SPA teams page first render was 96% quick than the traditional app. Across the board on the teams page the SPA is faster than the traditional app, it also has a speed index score almost half of its counterpart. The SPA is even visually complete before the traditional app has started rendering any content to the screen.

Appendix Fig 16.11 is the filmstrip showing frames of the rendering process of the teams page at 0.1 second intervals on a cached page load.
This film strip is a great example of the perceived load time advantages of SPA over traditional app. Even though the SPA doesn’t render the list of teams to the screen until 1.0 seconds there has been content on the screen for 900 milliseconds before that. The speed index score clearly shows the possibility of perceived load advantages of SPAs.

By comparing the two film strips it is clear that the SPAs only loads the list of teams onto the page 100 milliseconds before the traditional app. However, the single-page app has been User Ready [9] a whole 700 milliseconds before the traditional app — a user can interact with the filters to change the page content on the single-page app even before the entire page is fully loaded.

As single-page apps decouple data and presentation they can render to the screen components that have no data dependencies. Components like the header bar, filters, titles etc are all ready to the screen from 0.5 seconds. Traditional apps, however, bundle the data and presentation together. This coupling means that traditional app clients must wait for the longest possible task to finish before they can render anything to the page.

FIG 16.12 (below) is represents this property very clearly in the visual progress graph. The SPA gradually adds content to the page as soon as possible. The traditional app waits until it is almost all ready before rendering all of it to the screen.
Interaction Loading Times

The last two sections have looked at the initial page load when the client’s cache is empty and full. However, SPAs don’t navigate from page to page in the same way as traditional web apps. SPAs alter the DOM using JavaScript to display new content. Here we compare how traditional apps and SPAs handle navigating between pages, filtering lists of data and pagination.

In the case of the SPA we present the numbers from the JavaScript event that starts the JavaScript execution. Start render is the time to the first browser paint event and visually complete is the time from the start event to the last paint event associated with that start event.

**TEAMS PAGE: FILTER BY UNIVERSITY**

<table>
<thead>
<tr>
<th></th>
<th>Doc Complete Time (ms)</th>
<th>Fully Loaded Time (ms)</th>
<th>Start Render (ms)</th>
<th>Visually Complete (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>1152</td>
<td>1181</td>
<td>1197</td>
<td>1400</td>
</tr>
<tr>
<td>SPA</td>
<td>0</td>
<td>450</td>
<td>160</td>
<td>660</td>
</tr>
<tr>
<td>Difference</td>
<td>-1152</td>
<td>-731</td>
<td>-1037</td>
<td>-740</td>
</tr>
</tbody>
</table>

SPA first render 86% quicker; visually complete 52% quicker.

**TEAMS PAGE: LOAD MORE**

<table>
<thead>
<tr>
<th></th>
<th>Doc Complete Time (ms)</th>
<th>Fully Loaded Time (ms)</th>
<th>Start Render (ms)</th>
<th>Visually Complete (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>1152</td>
<td>1181</td>
<td>1197</td>
<td>1400</td>
</tr>
<tr>
<td>SPA</td>
<td>0</td>
<td>230</td>
<td>210</td>
<td>250</td>
</tr>
<tr>
<td>Difference</td>
<td>-1152</td>
<td>-951</td>
<td>-987</td>
<td>-1150</td>
</tr>
</tbody>
</table>

SPA first render 82% quicker; visually complete 82% quicker.

**NAVIGATE HOME TO TEAMS LISTING PAGE**

<table>
<thead>
<tr>
<th></th>
<th>Doc Complete Time (ms)</th>
<th>Fully Loaded Time (ms)</th>
<th>Start Render (ms)</th>
<th>Visually Complete (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>1152</td>
<td>1181</td>
<td>1197</td>
<td>1400</td>
</tr>
<tr>
<td>SPA</td>
<td>0</td>
<td>287</td>
<td>200</td>
<td>310</td>
</tr>
<tr>
<td>Difference</td>
<td>-1152</td>
<td>-894</td>
<td>-997</td>
<td>-1090</td>
</tr>
</tbody>
</table>
It is these test cases where SPA really start to shine over their traditional web app counterparts. SPAs are designed for adaptively changing the web page content. Functionality like filtering is perfectly suited to SPA design as this functionality alters only portions of the page, not all of it. SPAs clear only the teams that don’t match the filter locally, while in the background fetching the new list of teams. Then the SPA client renders the new teams to the screen.

Even in the case of navigating from the home page to the teams page it is clear that the SPA is better. SPAs skip the request, response, parsing steps of a page load.

Summary of Results

The results of these tests provide some interesting insights into the practical differences between traditional web apps and SPAs. On the uncached page load it is clear that the traditional app is faster. This is because the SPA client is waiting on the JavaScript bundle to be downloaded and parsed before it can do anything useful.

On the cached page load it was clear that the SPA can produce results that were either on par with traditional apps or in the case of the teams page produced numbers that blow traditional apps out of the water. The SPA could render content 96% faster to the screen than traditional apps. This is a big perceived performance difference for users. When we compared user interactions to the website like filtering or pagination it was immediately clear that the SPA is far superior to the traditional app.

Which design best suits a website depends on that website’s use case. Take for example a blog where users typically visit one article then leaves without interacting much with the site. This use case is better suited to the traditional web app as users will arrived with an empty cache and don’t need the advantages of the SPA's speedy interactions. If however, you’re use case is a web application where the user is actively performing actions then the speed advantages make a single-page app the obvious choice.
Part E: The Boilerplate Project

Section 17: Motivation
The primary motivation for this boilerplate project is to help developers skip the often complex setup time that is common when installing the framework from Section 8 and setting up the tools from Section 10.

As we’ve already pointed out that single-page app projects can be quite complex as you are developing in three languages to begin with, using multiple frameworks and libraries so that you’re not re-inventing the wheel on every project.

We spent 3 weeks setting up the underlying tooling and infrastructure for the JailbreakHQ project. This included setting up the project or development and deployment. In Section 10 we mentioned the large number of tools required to build complex single-page apps. In Section 12 we discussed a number of grunt tasks that convert the easy to debug development project into one that could be deployed into production.

The boilerplate project aims to provide you with the scaffolding to get you building and deploying single-page apps much faster than starting out on your own. The boilerplate project illustrates how it works with a small set of example patterns. These patterns provide a single-page app that displays information about the boilerplate project’s commit and issue history on Github using Github’s API.

Every single-page app needs a small series of extra concerns. Those include adding a 404 page not found view, a framework for preventing memory leaks from old views etc. These universal features of all single-page apps are also included in the boilerplate project.

The design process for the boilerplate project was that we took the JailbreakHQ project and abstracted out the pieces that were not specific to JailbreakHQ. Every component of the boilerplate project has a use and was used to build JailbreakHQ,
which has been battle-tested by over 15,000 users. By extracting the boilerplate project from a concrete example like JailbreakHQ we have reduced the possibility of over-designing this boilerplate with unnecessary tools and features.

The boilerplate project includes all of the tools outlined in Section 10 including bower, require.js, grunt; the intermediary languages such as CoffeeScript, SASS and Jade and we have configured the Frameworks Backbone and Foundation. All of these tools and frameworks provide you with a solid base to build your single-page app on top of.
Section 18: Getting Started

Getting the Project Running Locally

The first step is to clone the boilerplate project’s GitHub repo 27 so that you have a local copy of the boilerplate code. You should then install the following dependencies into your system.

```
node
npm
bower
```

All of these are available through the brew package manager for Mac and the apt package manager for unix machines. Note that when using the apt package manager the node package is called *nodejs* not *node*.

**DEPENDENCIES**
Once you have those dependencies installed execute the following script in the root of the project.

```
./setup.sh
```

This script will install the grunt packages, install project dependencies using bower like Foundation, jQuery and Backbone, and build the project initially using the Grunt. This will convert the provided CoffeeScript, SASS and Jade files in ‘/static/src’ into the corresponding JavaScript, CSS and HTML documents and place them in the directory ‘/static/build’.

**SERVING THE PROJECT LOCALLY**
When you install node it also install a very lightweight command-line HTTP server. Run the following command inside the ‘/static/’ directory:

```
http-server -c-1
```

---

27 Boilerplate Project Repo; https://github.com/Specialkbyte/boilerplate-single-page-app
You are now running a local HTTP server (without any caching enabled - useful for developing) at http://localhost:8080. Open that URL in your browser and you should see the boilerplate single-page app.

Developing the Project

Now that you have the project running locally you just need to know which files to edit and how to rebuild locally.

WHICH FILES DO YOU EDIT?
You should only edit the files inside the ‘/static/src/’ directory. This is where the coffeescript, SASS and jade files are located. You do not want to edit the files inside the ‘/static/build/’ directory. These files are automatically generated by the grunt task runner.

You can install front-end dependencies using bower. These are installed into the ‘/static/components/’ directory. You shouldn't edit code in here either. As this directory is build with a clean set of packages on every deploy build.

**USING THE GRUNT FILEWATCHER**
This project is setup to run Grunt to build the intermediary languages like coffeescript and SASS into javascript and CSS. In a terminal window run and leave running while developing:

```plaintext
grunt watch
```

This starts the Grunt filewatcher that will generate the output of the corresponding source files to your CoffeeScript, SASS, and Jade files whenever you change them. If you forget to run this command and leave it running in the background whenever you make file changes to your CoffeeScript, SASS, Jade files it will have no affect on the actual project.

Now that you have the project running locally and you know how to edit project files we suggest you have a look around the structure of the project. Discover how the source files are arranged, what effect does changing them have, etc.

**Deploying Your Project**
You need to make some minor alternations to the project in order to deploy it. This is because everyones deployment process will be slightly different. In the ‘/static/src/html/index.jade’ you need to replace ‘//builds.yourhost.com…’ with the hostname of the server that will eventually serve your static file contents.

The next steps are the recommended way of deploying the project. They are not 100% prescriptive. If you haven’t already you need to create a GitHub repo for your
project, signup to TravisCI and signup to Amazon AWS in order to create an S3 Bucket.

Once you have done that get your project to build on TravisCI. The boilerplate project comes with a TravisCI config called ‘.travis.yml’ in the project root. This config tells TravisCI how to install dependencies and build the project using Grunt.

You should then setup an IAM access role your Amazon S3 Bucket. Google online for guides on how to best setup those credentials.

You will want to add deploy directives to the TravisCI config file. This will tell Travis what to do with your build files after building them. We want to upload them to amazon S3. Using your Amazon S3 Bucket IAM credentials use the TravisCI command line tool to generate encrypted copies of those credentials that you can place in your .travis.yml. There is an example of the config likes you want to add in the JailbreakHQ TravisCI config file.

```yaml
deploy:
  - provider: s3
    skip_cleanup: true
    access_key_id: AKIAIX6A5B5HPFKUJTCA
    secret_access_key:
      secure: Etm5dc+29wTOY1o9XX6Bg0rAD6Lb57nqpOgLEju+jhPQ3uRMYdbQepzSLjBVIDPswqz7L1oIILhQZ3Pzx6ShB7cDLu9+LQw7mIC0ErZvmXujk06y6hp7RFaMtws8Qa/cmcahKgyFPyD6cqBPK0gOI2feIx6LwtoUEGL6x2rQdco=
    bucket: builds.jailbreakhq.org
    region: eu-west-1
    endpoint: s3-eu-west-1.amazonaws.com
    local-dir: artifact
    upload-dir: static-${TRAVIS_BUILD_NUMBER}
    acl: public_read
    on:
      repo: jailbreakhq/public-static
      branch: master
```

For example this config section tells TravisCI to use the encrypted credentials to upload to the builds.jailbreakhq.org Amazon S3 bucket. That bucket is in the Amazon

---

28 JailbreakHQ Travis Deploy Config: https://github.com/jailbreakhq/public-static/blob/master/.travis.yml#L17-45
eu-west-1 region. We want to upload the locally built directory ‘artifact’ to the
directory ‘static-${TRAVIS_BUILD_NUMBER}’. That build number is a placeholder
which is replaced with the TravisCI build number. This is what creates the uniquely
numbered release numbers we discussed in Section 12. The ‘acl’ directive tells
Amazon S3 that these files should be publicly available over the internet. The final
‘on’ directive tells TravisCI that this deploy command should only be run whenever
there is a push to the master branch of the JailbreakHQ GitHub repo.

After you have setup this up make a trivial change and attempt to deploy your
project. Check that the files do exist in your Amazon S3 bucket in the appropriate
locations by using the Amazon S3 management console.

After this point you may need to config DNS records to provide a pretty host name to
your S3, setup a CDN like Cloudflare etc. This is outside the scope of the project but
Section 7 contains some useful advice.
Section 19: Useful Patterns

The boilerplate project provides many useful patterns for developing Backbone single-page apps. The JailbreakHQ project provides even more complex and advanced patterns.

Composing Backbone Views

NESTING VIEWS

A useful pattern presented in the boilerplate project is nesting Backbone views. The IndexView of the boilerplate project nests two views inside it. The CommitsListView and the IssuesListsView.

FIG 19.1 The red outline shows the IndexView render area. The yellow outline shows the CommitsListView nested inside the IndexView. The green outline shows the IssuesListview nested inside the IndexView.
Each view should manage one piece of state information as we discussed in Sections 2 and 3. This means that when the piece of state information changes only one view must re-render in order to update the display. If one view was based on many pieces of state information we would be required to re-render the everything not just one component of the website.

One of the key pieces of this pattern is that the IndexView, the parent view, creates the CommitsListView using the collection ‘@commits’. The IndexView then appends that view to the DOM element with the id attribute ‘commits’. The IndexView template contains a HTML div tag with the id attribute ‘commits’. It is important to point out that the jQuery reference in the last line uses the DOM tree specifier ‘@$el’. That tells jQuery to look for the element with the id attribute ‘commits’ inside this view’s element attribute and not in the document DOM tree. This is important because the view’s element might not have been rendered to the document DOM tree when we make this call. Not using this DOM tree specifier will lead your single-page app to render the CommitsListView to the user only some of the time.

LIST PATTERN
One other common Backbone pattern which is illustrated in the boilerplate project is the list pattern. The CommitListView is actually a parent view, which create a CommitItemView foreach commit in the collection and appends those child views inside the CommitListView.

```javascript
@commitsView = new CommitsListView
    collection: @commits
...

$('#commits', @$el).append @commitsView.render().$el
```

```javascript
render: =>
   @$el.html @template @getRenderContext()
   _.each @collection.models, (commit) =>
       commitView = new CommitItemView
       model: commit
       @rememberView commitView
       $('#commits-list').append commitView.render().$el
```
The render function for the CommitsListView iterates over the collections, create a CommitItemView for each model in the collection. It then renders and appends that commitView to the div with id attribute ‘commits-list’ which is defined inside the CommitsListView template.

HubSpot Mixen Library Examples

In Section 10 we said we would provide examples of usages of the Mixen Library for creating compassable and re-usable chunks of JavaScript code. Here we provide two examples: Event Janitor Mixen which closes Backbone views after we are done with them in order to prevent memory leaks; and the Syncing Mixen which keeps track of the syncing state of the collection or model for a view. If you haven't read the Mixen documentation\textsuperscript{29} we highly recommend you do that now.

EVENT JANITOR

When we navigate from page to page in a single-page app we want to close the old views and tidy-up after ourselves. This requires developers to remove JavaScript DOM listeners as this can prevent the JavaScript garbage collector from cleaning up old views. When we close a view we also want to close that view's child views if it has any. The EventJanitor is a pattern for managing this tidy-up process.

```javascript
class EventJanitor
    rememberView: (view) ->
      @views ?= []
      @views.push view
      view

    close: ->
      @stopListening()
      if @views?
        for view in @views
          if view?.close
```

**FIG 19.4** The EventJanitor Mixen class demonstrating use of the Backbone’s stopListening() and remove() method

\textsuperscript{29} HubSpot Mixen Documentation; \url{http://github.hubspot.com/mixen/}
We have mixed this EventJanitor Mixen into the BaseViewMixen, which is what every backbone view in the boilerplate extends. Whenever we create a new nested view we can then call the `@rememberView(view)` function in the parent view so that it can remember to close it if needs be.

LOADING AND ERROR STATES
Another pattern that we found was common when building the *JailbreakHQ* project was that of managing loading and error states for views. Each view is provide with a state object (collection, model). The single-page app makes AJAX requests to the API server when we call the `fetch` method on those state objects. During the phase when we are loading the data from the API server it would be nice to show the user a loading message or spinner. If the API server returns an error it would be convenient to display an appropriate error message to the user as well.

Inside the issues.jade template there is an example of how this pattern manifests in the template (FIG 19.5). If there is an error display an error message to the user. If we are still loading the model/collection display the loading icon. Otherwise present the users with the list of issues.

```
| h3 Issues
| if error
|     | Error loading issues
| else if loading
|     include loading.jade
| else
|     ul#issues-list
```

FIG 19.5 The loading/error state pattern in the issues.jade template

The Syncing Mixen hooks into the Backbone sync function and listens for Backbone events and sets appropriate flags on the model/collection. The view can then insert those model/collection state flags into the template context in the render function. This pattern is easily achieved using the BaseCollectionMixen and BaseModelMixen for collections and models respectively. Use the CollectionViewMixen and ModelViewMixen to automatically include the those state flags inside your view’s render context. Then use pattern in 19.5 and you will have loading error states for users.
Section 20: Further Work

There is more opportunity for growth within the boilerplate project. There are yet unfinished features and new areas to explore like testing.

TESTING
The boilerplate project provides no JavaScript testing framework out-of-the-box. This is a glaring omission but was necessary in order to fulfill the project deadline. There are many types of JavaScript testing from unit testing, to behavioural testing. JavaScript as a language can be quite hard to test because of its deep coupling with the DOM [28].

INTERNATIONALISATION
It would also be useful if the boilerplate project included an internationalisation framework example such as i18n.js. Internationalisation is a common problem for web app and sometimes a needlessly complex task to setup. Bundling internationalisation could remove yet another complex setup task from developers.

DEPLOY ROLLBACKS
It would be useful if the deploy rollbacks feature of this boilerplate project was completed as discussed in Section 12.
Section 21: Concluding Remarks

In Section 2 we explained the pitfalls and problems faced by traditional and Web 2.0 web apps. We explained what are single-page apps and how they work by moving all rendering logic to the client. In Section 3 we provided a philosophy for building manageable single-page apps that encourages separation of concerns and loose coupling. In Section 4 we explained the advantages of single-page apps for users such as faster loading times, smoother user experience. We also discussed the architectural advantages like separation of data and presentation, the ability to make your website another consumer of your single API and the ability for single-page apps to remix data from multiple data sources all on the client.

In Section 6 we explained how the example project JailbreakHQ presents some interesting challenges including authentication and SSL everywhere to our development team. In Section 7 we presented a possible deployment layout that leverages modern cheap cloud technologies that fosters speed and scalability. In Section 8 we choose BackboneJS as our single-page app framework as it best matches the design philosophy from Section 3. We also choose to include Foundation CSS as it provides many mobile friendly features.

In Section 10 we presented a long list of tools that are a necessary evil to manage the complexities of single-page app projects. These included bower for dependency management, require.js for JavaScript modules, Grunt as a task runner. We also presented the advantages of using CoffeeScript, SASS and Jade over their counterparts JavaScript, CSS and HTML. In Section 12 we briefly describe the goals of the boilerplate and JailbreakHQ deployment pipeline and how it optimises static files for delivery to end users. In Section 13 we briefly touched on possible tools you could add to your single-page app for monitoring your single-page app. These tools are not included by default in the boilerplate project.
In Section 15 we compared the data volume transfer of a test traditional web versus a functionally similar single-page app. We found an increase the JavaScript payload for single-page apps but as a percentage of the total data volume transfer the increase was insignificant. In Section 16 we compared the actual and perceived load times for the two web app styles. Traditional apps we generally faster on a cold cache, with single-page apps generally faster on a hot cache. Single-page apps were as much as 85% faster than their traditional counterparts on actions like single-page app navigation and filtering/pagination.

Finally in Part E we presented the boilerplate project that included all of the tools, and frameworks from Sections 8 and 10, and the majority of the deployment pipeline from Section 12. We hope that the boilerplate project, as it is based on the abstract sections of a real-world project like JailbreakHQ will be valuable to other developers of single-page apps.

There is no where near enough time and pages of paper to explain all of the features and patterns in the JailbreakHQ project and boilerplate project. Hopefully we have touched upon the core ideas that will allow you build single-page applications using the boilerplate project.
Appendix

Fig 4.2 Facebook Home Page with Placeholder Content ii
Fig 4.3 JailbreakHq.org Home Page Partially Loaded ii
Fig 16.1 W3C Navigation Timing Diagram iii
Fig 16.2 Home Initial Load Filmstrip iv
Fig 16.3 Home Initial Load Visual Progress v
Fig 16.4 Home Initial Load Timing Events v
Fig 16.5 Teams Initial Load FilmStrip vi
Fig 16.6 Teams Initial Load Visual Progress Graph vii
Fig 16.7 Teams Initial Load Timing Events vii
Fig 16.8 Home Cached Filmstrip viii
Fig 16.9 Home Cached Visual Progress ix
Fig 16.10 Home Cached Timing Events ix
Fig 16.11 Teams Cached FilmStrip x
Fig 16.12 Teams Cached Visual Progress xi
Fig 16.13 Teams Cached Timing Events xi
FIG 4.2 FACEBOOK HOME PAGE WITH PLACEHOLDER CONTENT

FIG 4.3 JAILBREAKHQ.ORG HOME PAGE PARTIALLY LOADED
FIG 16.1 W3C NAVIGATION TIMING DIAGRAM
FIG 16.3 HOME INITIAL LOAD VISUAL PROGRESS

![Graph showing visual progress (%) for SPA Home and Trad Home]

FIG 16.4 HOME INITIAL LOAD TIMING EVENTS

![Bar chart showing timings (ms) for SPA Home and Trad Home (HTTP)]
FIG 16.5 TEAMS INITIAL LOAD FILMSTRIP

<table>
<thead>
<tr>
<th>Time</th>
<th>SPA</th>
<th>TRAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0s</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>0.5s</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>1.0s</td>
<td>56%</td>
<td>90%</td>
</tr>
<tr>
<td>1.5s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.0s</td>
<td>57%</td>
<td>97%</td>
</tr>
<tr>
<td>2.5s</td>
<td>71%</td>
<td></td>
</tr>
<tr>
<td>3.0s</td>
<td>78%</td>
<td></td>
</tr>
<tr>
<td>3.5s</td>
<td>100%</td>
<td></td>
</tr>
</tbody>
</table>
FIG 16.6 TEAMS INITIAL LOAD VISUAL PROGRESS GRAPH

![Visual Progress Graph](image)

FIG 16.7 TEAMS INITIAL LOAD TIMING EVENTS

![Timing Events](image)
FIG 16.9 HOME CACHED VISUAL PROGRESS

![Graph showing visual progress over time for SPA Home and Trad Home]

FIG 16.10 HOME CACHED TIMING EVENTS

![Bar chart showing timings for different events for SPA Home and Trad Home]
FIG 16.11 TEAMS CACHED FILMSTRIP

SPA

0.0s
0%

0.1s
19%

0.2s
19%

0.3s
19%

0.4s
19%

0.5s
57%

0.6s
57%

0.7s
57%

SPA

0.8s
57%

0.9s
57%

1.0s
57%

1.1s
100%

TRADE

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%

0%
FIG 16.12 TEAMS CACHED VISUAL PROGRESS

Visual Progress (%)

Time (seconds)

FIG 16.13 TEAMS CACHED TIMING EVENTS

Timings (ms)

Appendix


4. Tim O'Reilly; What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software (March 2007)


9. Mike Petrovich; User Ready Time (2013) (Online)


11. Folmer, E; Software Architecture analysis of Usability (2005)

12. Teo, Oh, Liu, Wei; An empirical study of the effects of interactivity on web user attitude (2003)


15. C Smullen III, A Smullen; An experimental study of AJAX application performance (March 2008)


18. David Herman, Sam Tobin-Hochstadt; Modules for JavaScript: Simple, Compilable, and Dynamic Libraries on the Web (2011)


20. S Souders; High-performance web sites (December 2008)


27. Douglas Crockford; Prototypal Inheritance in JavaScript (http://javascript.crockford.com/prototypal.html)


29. Devesh Tiwari and Yan Solihin; Architectural Characterization and Similarity Analysis of Sunspider and Google’s V8 Javascript Benchmarks

30. Douglas Crockford; JavaScript the Good Parts (Book)


32. Stackoverflow; How does JavaScript “this” work? (http://stackoverflow.com/questions/3127429/how-does-the-this-keyword-work)

33. Douglas Crockford; Classical Inheritance in JavaScript (http://javascript.crockford.com/inheritance.html)

34. Heroku; 12 Factor App, III Config (http://12factor.net/config)


36. Martin Fowler; Continuous Integration (http://www.dccia.ua.es/dccia/inf/assignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf)
37. Humble, Farley; Continuous Delivery: through build, test and deploy automation (Book) (http://ptgmedia.pearsoncmg.com/images/9780321601919/samplepages/0321601912.pdf)

38. Martin Fowler, Jim Highsmith; The Agile Manifesto (2001)


40. HTTPbis Working Group; HTTP/2 Specification Draft (https://http2.github.io/http2-spec/)

41. HTTPbis Working Group; HTTP/2 FAQ (https://http2.github.io/faq/)

42. Krasner and Pepe; Module, View Controller (1988)